Solving the k-Selection Problem

Parallel and Distributed Systems

Christina Koutsou
github: @kchristin22

1 About

The purpose of this project is to find the kth smallest element of a given vector that is too large to fit in one machine, thus making the use of MPI necessary.

2 Algorithms and Implementation details

All algorithms follow the principal of performing the same calculations instead of letting one node do the computations and broadcast the results. This tactic aims to minimize the communication between processes and avoid relying heavily on a single node of the cluster. A simple example present in the implementations is the use of `MPI_Allreduce` which also includes a broadcast of the final reduction, rather than letting only the master know the reduction output. Moreover, it is assumed that no additional space is available for auxiliary buffers.

2.1 k-Search

This implementation searches for the kth element by calculating a heuristic pivot based on the range of the elements in the array and then counting the number of elements that are less than or equal to it. In more detail, the local minimum and maximum values are recorded, reduced with the ones from the rest of the processes and broadcast. In case the vector contains only one value multiple times ($min = max$), the k element is returned at once. The same applies in case the first ($k = 1$) or the last element ($k = n$) is requested. Otherwise, the pivot is iteratively adjusted by performing linear interpolation to narrow down the range where the kth element resides by associating the two previous pivot values with the number of elements less than or equal to each one of them. The goal is to find the pivot value that corresponds to the aforementioned count being equal to k.

When choosing a pivot, it is assured that it stays between the range denoted by the overall min and max and that it is also different from the last two pivot values, in order to avoid looping between them. In the latter case, where the calculated pivot has been previously chosen, the pivot is differentiated from that value by adding or subtracting 1 based on whether the elements less than or equal to that previous pivot were less than k or not.

The program settles in a final pivot if:

1. the two latter pivots are 1 integer apart and k is in the range denoted by the corresponding counts of smaller or equal elements. This signifies that at least the pivot value with the greater count of elements is in the array and is in fact the kth element.

2. The count is equal to k. Since the pivot's value may not be present in the array, the element with the smallest negative or zero distance from the pivot value (less than or equal to the pivot) is the kth element.

3. The count is equal to k - 1. The element with the closest positive distance from the pivot value is the kth element.

Otherwise, if none of the above conditions are met and the count for the current pivot is greater than k, the elements that are larger than the pivot are erased. In addition if the count becomes small enough in the process, the local arrays are gathered to a single node to continue computations.

It is noteworthy that the erasure of the elements may not be desirable in some applications and, thus, the program can be altered to exclude that part. However, if reducing computation time is our ultimate goal and, hence, gathering the elements to a single node is preferable, we would eventually erase at least the master's elements greater than the current pivot in order to fit the incoming elements from the other processes. These implementation details are to be tailored to the specific needs of the application concerning the execution time and the distribution of data.

2.2 Heuristic Quickselect

This implementation serves as the bridge between the k-Search and Quickselect algorithms. The process of choosing a pivot, and thus the overall program, resembles the k-Search version. However, instead of simply counting the elements less than or equal to the pivot, they are being separated from the larger values at every iteration and their number is returned (the pointer to the last element position of the left sub-array). Hence, the range of searching is decreasing as the program progresses, without having to erase any elements. The swapping of the elements is performed by searching simultaneously from both directions of the array: starting from the left margin, the search is stopped as soon as an element greater than the pivot is found, while the opposite holds for the other direction. When both iterators exit from their respective loop, they either point to elements that need to switch places or point to the same position, signifying that the array is partitioned successfully.

A minor addition to the Quickselect algorithm is further limiting the array being succumbed to internal swaps, by using past pointers as indications of sub-arrays being knowingly less or larger than future pivot values and hence needing no further checking. The following example clarifies this notion:

Consider the array: 8 8 2 8 9 8 9 4. In the first iteration of searching for the 4th element, the pivot is computed as:

$$p = max + \frac{(k-n)(max-min)}{n} = 9 + \frac{(4-8)(9-2)}{8} = 9-3 = 6$$

After partitioning the array, it is now: 4 2 8 8 9 8 9 8, and the count of elements ≤ 6 is equal to 2. Since 2 is neither equal to 4 nor $4-1=3$, a new pivot must be calculated which should be larger than the previous one due to linear interpolation. As it is known that the elements in positions 0 and 1 are already less than or equal to 6 and the next pivot will be larger, the swapping process can begin from position 3. This poses no problem to the algorithm since the elements are not counted like in the case of k-Search, but rather a pointer is returned.

The narrowing of the array in question also applies to the part where the final pivot is found and the array is searched for the closest element.

2.3 Quickselect

In the Quickselect algorithm the pivot value is randomly chosen by the master from its local array, which is afterwards partitioned based on its value. Since the partitioning is narrowing the array of interest, the master's local array may not include any elements in that range. Furthermore, the last two pivot values should not be repeated to avoid looping between them. Consequently, an alternation of the master process is performed when necessary. If none of the processes satisfy these

conditions, then the only elements left are equal to at least one of the two previous pivots and hence the kth element is one of them and its value is determined by the following checks:

```
if (p == prevPrevP || p == prevP)
    {
        if (std::clamp(k, prevCountSum, countSum) == k || std::clamp(k,
            countSum, prevCountSum) == k) // k is between the two countSums
            (prevCountSum corresponds to the prevPrevP)
            kth = std::max(prevP, prevPrevP); // the max value from the two is
                chosen
        else if (countSum > k) // both counts are larger than k
            kth = std::min(prevP, prevPrevP); // smallest pivot value
                corresponds to smallest countSum
        else // both are smaller than k
            kth = std::max(prevP, prevPrevP);

        return;
    }
```

3 Benchmarks

To perform the benchmarks the `nanobench` tool was used. Only the output of the process with ID = 0 is displayed, as it is the one responsible to carry out the rest of computations when the array is small enough to be gathered. Each version is run for 5 epochs with at least 10 iterations each to stabilize the results. The number of iterations and epochs can be fine-tuned according to the median error percentage.

Before each iteration the input arrays are re-initialized. This step is also timed in order to find the actual time of the program executions. Moreover, for the Quickselect implementations the sequential version of partitioning was used as the nature of its algorithm favors it over the parallel one. As far as randomness is concerned, the current Quickselect implementation truly only uses it when selecting the first pivot, but to further decrease this parameter's influence in the results, the seed is the same in every run of the program.

For the tests, Aristotle HPC was used and specifically its batch partition with maximum 5 nodes per user. The dataset used was a wikipedia archive dump.

Firstly, the effect of the number of nodes used is measured. Each node of the cluster partition executes a single task. This approach is preferable over having less physical nodes with more tasks each, as memory and local parallelism are more efficiently managed, while the number of MPI nodes is not altered. The comparison of the two options using their median value is shown in Table 3.1.

	N = 4 & ntasks=per-node = 1	**N = 2 & ntasks=per-node = 2**
k-Search	100.179.873 ns	120.482.716 ns
Heuristic Quickselect	48.884.716 ns	87.290.836 ns
Quickselect	86.286.447 ns	77.360.999 ns

Table 3.1: *Comparison of distribution of 4 MPI processes (k = n / 2)*

Nodes	k-Search	Heuristic Quickselect	Quickselect
1	291.030.440	173.345.003	319.248.596
2	178.450.248	96.490.866	154.899.992
3	124.907.730	60.404.277	108.077.973
4	100.179.873	48.884.716	86.286.447
5	85.839.037	45.117.236	77.346.095

Table 3.2: *Number of MPI processes (k = n / 2): median values in ns*

It is evident that when we increase the number of processes, the performance improves due to the division of the workload and of the array, as more and more of a process' local array range fits in the machine's cache memory. Unfortunately, the batch partition imposes a constraint of no more than 5 nodes per user, thus the optimal number of nodes— beyond which performance begins to decline— could not be determined for this dataset.

k	k-Search	Heuristic Quickselect	Quickselect
1	4.980.477	4.844.976	36.987.014
2	20.018.794	10.932.261	40.536.907
n / 4 = 7021089	53.831.297	36.717.278	71.138.04
n / 2 = 14042179	97.295.391	46.345.860	77.111.490
3n / 4 = 21063269	43.543.126	34.594.964	60.822.740
n - 1 = 28084358	20.508.361	16.382.804	45.754.562
n = 28084359	4.873.906	4.668.849	45.099.328

Table 3.3: *Median execution times in ns for different k values*

The relationship between the parameter k and the corresponding execution time is illustrated in Table 3.3. Notably, the execution time reaches its peak when k = n / 2, gradually diminishing as k approaches the array's ends. The balanced nature of the results is attributed to two factors: firstly, the broader search range for the kth element when k = n / 2, likely due to pivot selection or the data distribution; and secondly, the uneven distribution of workload. Furthermore, in k-Search, when k >= n / 2, the algorithm counts elements larger than the pivot, reinforcing similarities in results for k values equidistant from the nearest end of the array. Specifically for k = 1 and k = n, k-Search and Heuristic Quickselect leverage the upfront identification of overall minimum and maximum values, enabling faster completion compared to Quickselect and subsequent k values (e.g., k = 2 and k = n - 1).

Finally, it is noteworthy that the Heuristic Quickselect outperforms the other methods, offering up to 2x speedup to k-Search and up to 3x to Quickselect (general case). It is evident from the first comparison that the reduction of the search range using prior outcomes from different pivots, as explained in Section 2.2, has a big impact on the algorithm. In addition, the second comparison indicates that the selection of the pivot plays an even more significant role in the performance of the algorithm.

For more information about the statistics of all the above measurements refer to the README.